Programmation Fonctionnelle Avancée

Séance 8 : Monade de listes, MonadPlus

Alexandros Singh

Université Paris 8

1er décembre 2023

Une nouvelle monade de listes

- Nous avons déjà vu que les listes forment un monoïde et qu'elles peuvent donc être utilisées pour construire une monade Reader qui garde une trace des calculs avec une liste.

Une nouvelle monade de listes

- Nous avons déjà vu que les listes forment un monoïde et qu'elles peuvent donc être utilisées pour construire une monade Reader qui garde une trace des calculs avec une liste.
- Aujourd'hui, nous allons voir comment le type liste lui-même forme une monade, qui peut être utilisée pour représenter des calculs qui renvoient zéro ou plusieurs résultats.

Une nouvelle monade de listes

- Nous avons déjà vu que les listes forment un monoïde et qu'elles peuvent donc être utilisées pour construire une monade Reader qui garde une trace des calculs avec une liste.
- Aujourd'hui, nous allons voir comment le type liste lui-même forme une monade, qui peut être utilisée pour représenter des calculs qui renvoient zéro ou plusieurs résultats.
- Cette monade est souvent utilisée pour modéliser des calculs "non déterministes" en programmation.

Une nouvelle monade de listes

- Nous avons déjà vu que les listes forment un monoïde et qu'elles peuvent donc être utilisées pour construire une monade Reader qui garde une trace des calculs avec une liste.
- Aujourd'hui, nous allons voir comment le type liste lui-même forme une monade, qui peut être utilisée pour représenter des calculs qui renvoient zéro ou plusieurs résultats.
- Cette monade est souvent utilisée pour modéliser des calculs "non déterministes" en programmation.
- Pour ce faire, les calculs "non déterministes" renvoient des listes de résultats possibles.

Définition de la monade de listes

Définissons maintenant la monade de liste :

```
1 module ListMonad : Monad with type 'a t = 'a list = struct
2     type 'a t = 'a list
```

Le type monadique est bien sûr le type des listes !
Et `return` ? Qu'est-ce qu'une "boîte standard" ici ?

Définition de la monade de listes

Définissons maintenant la monade de liste :

```
1 module ListMonad : Monad with type 'a t = 'a list = struct
2     type 'a t = 'a list
3     let return x = [x]
```

Et `>>= : 'a t -> ('a -> 'b t) -> 'b t` ?

- On nous donne une valeur monadique `x : 'a list`
- et une fonction `f : 'a -> 'b list`,
- comment appliquer `f` aux éléments de `x` et obtenir une seule liste à la fin ?

Définition de la monade de listes

Définissons maintenant la monade de liste :

```
1 module ListMonad : Monad with type 'a t = 'a list = struct
2     type 'a t = 'a list
3     let return x = [x]
4     let ( >>= ) x f = List.concat (List.map f x)
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= return;;
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= return;;
- : int list = [1; 2; 3; 4]
```

Cela découle des lois sur les monades - vérifiez également les deux autres lois !

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x+1]);;
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x+1]);;
- : int list = [2; 3; 4; 5]
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x+1]);;
- : int list = [2; 3; 4; 5]
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x+1]) >>= (fun x -> [x*2]);;
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x+1]) >>= (fun x -> [x*2]);;
- : int list = [4; 6; 8; 10]
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x;x]);;
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x;x]);;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4]
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x;x]) >>= (fun x -> [x;x]);;
```

Exemples avec la monade de listes

Testons notre nouvelle monade ! Essayez de deviner à quoi s'évalue l'expression suivante :

```
# [1;2;3;4] >>= (fun x -> [x;x]) >>= (fun x -> [x;x]);;
- : int list = [1; 1; 1; 1; 2; 2; 2; 2; 3; 3; 3; 3; 4; 4; 4; 4]
```

En poursuivant notre discussion sur les "calculs non déterministes", nous aimerions que nos modèles comportent également les deux éléments suivants :

En poursuivant notre discussion sur les "calculs non déterministes", nous aimerions que nos modèles comportent également les deux éléments suivants :

- Une valeur qui représente l'échec d'un calcul (une branche morte si nous considérons les calculs non déterministes comme des arbres).

En poursuivant notre discussion sur les "calculs non déterministes", nous aimerions que nos modèles comportent également les deux éléments suivants :

- Une valeur qui représente l'échec d'un calcul (une branche morte si nous considérons les calculs non déterministes comme des arbres).
- Un moyen de faire des choix entre les résultats possibles et de les fusionner.

En enrichissant la signature des monades avec ces deux éléments, nous obtenons :

```
module type MonadPlus = sig
    include Monad
    val mzero : 'a t
    val mplus : 'a t -> 'a t -> 'a t
end
```

L'ensemble précis des lois satisfaites par MonadPlus varie. Il contient généralement certaines des lois suivantes :

```
mplus mzero a = a
mplus a mzero = a
mplus (mplus a b) c = mplus a (mplus b c)
mzero >>= k = mzero
mplus a b >>= k = mplus (a >>= k) (b >>= k)
mplus (return a) b = return a
```

Nous pouvons maintenant créer une structure utilisant des listes qui a la signature de MonadPlus.

- Les calculs qui échouent sont représentés par des listes vides :

```
module ListMonadPlus : MonadPlus with type 'a t = 'a list = struct
   include ListMonad
   let mzero = []
```

Nous pouvons maintenant créer une structure utilisant des listes qui a la signature de MonadPlus.

- Les calculs qui échouent sont représentés par des listes vides :

```
module ListMonadPlus : MonadPlus with type 'a t = 'a list = struct
   include ListMonad
   let mzero = []
```

- Le fusionnement se fait par append :

```
   let mplus = List.append
end
```

Pour tester notre compréhension :

```
# mplus [1;2;3] mzero;;
```

Pour tester notre compréhension :

```
# mplus [1;2;3] mzero;;
- : int list = [1; 2; 3]
```

Pour tester notre compréhension :

```
# mplus mzero [1;2;3];;
- : int list = [1; 2; 3]
```

Pour tester notre compréhension :

```
# [1;2;3] >>= (fun x ->
    mplus ((fun x -> [x+1]) x)
          ((fun x -> [x-1]) x));;
```

Pour tester notre compréhension :

```
# [1;2;3] >>= (fun x ->
    mplus ((fun x -> [x+1]) x)
          ((fun x -> [x-1]) x));;
- : int list = [2; 0; 3; 1; 4; 2]
```

En utilisant `let*` et un opérateur `<|>` inline pour `mplus`, cela devient plus propre :

```
let (let*) x f = x >>= f
let ( <|> ) = mplus
let example2 =
   let* a = [1;2;3;4] in
   (fun x -> [x+1]) a <|> (fun x -> [x-1]) a
```

MonadPlus nous permet de faire d'autres choses intéressantes. Par exemple, la fonction suivante force certains des calculs alternatifs possibles à échouer sur la base d'un prédicat :

```
let guard : bool -> unit t = function
    | true -> return ()
    | false -> mzero
```

Essayez de deviner le résultat :

```
# let* a = [1;2;3;4] in
  let* _ = (a >= 3) |> guard in
  return a
```

Essayez de deviner le résultat :

```
# let* a = [1;2;3;4] in
  let* _ = (a >= 3) |> guard in
  return a
- : int list = [3; 4]
```

Pourquoi ? Réfléchissez aux deux choses suivantes :

- L'expression se réécrit comme :

```
# [1;2;3;4] >>= fun a -> ((a >= 3) |> guard >>= fun _ -> return a);;
- : int list = [3; 4]
```

- Quel est le résultat de l'application d'une fonction constante à `()` ?

```
# [(); (); ()] >>= fun _ -> return 5;;
- : int list = [5; 5; 5]
```

Option et MonadPlus

Enfin, la monade Maybe s'étend naturellement à une MonadPlus :

```
module MaybeMonadPlus : MonadPlus with type 'a t = 'a option = struct
    type 'a t = 'a option
    let return x = Some x
    let ( >>= ) x f = match x with
        | None -> None
        | Some x -> f x
    let mzero = None
    let mplus x y = match (x,y) with
    | None, None -> None
    | Some x, None -> Some x
    | None, Some x -> Some x
    | Some x, Some _ -> Some x
end
```

(Attention : les instances MonadPlus de `'a list` et `'a option` satisfont chacune à certaines lois que l'autre ne satisfait pas !)